IN THE UNITED STATES DISTRICT COURT

FOR THE NORTHERN DISTRICT OF CALIFORNIA

ORACLE AMERICA, INC.,

Plaintiff,

v.

GOOGLE INC.,

Defendant.
_____/

No. C 10-03561 WHA

CLAIM CONSTRUCTION ORDER

INTRODUCTION

In this patent and copyright infringement action involving computer technology, the parties seek construction of six terms found in six of the seven asserted patents. On April 27, 2011, a tentative claim construction order was issued, and the parties were invited to file five-page critiques of the constructions therein. After consideration of the supplemental briefing from both sides, final constructions for five of the six terms are set forth below. The sixth term, which is really at least three separate terms, will not be construed at this time.

STATEMENT

The Java software platform is central to this action. The patents at issue claim improvements to the Java technology, so a basic understanding of Java is a necessary foundation for construing the disputed patent terms. The Java concepts explained in this portion of the order are *not* part of the claimed inventions of any asserted patents; rather, they constitute the prior art upon which those inventions built.

Java was (and is) a technology that enabled software developers to write programs that were able to run on a variety of different types of computer hardware without having to rewrite the programs for each different type of computer hardware or "machine." Java was not the first "write once, run anywhere" solution, but it has become one of the most popular software platforms.

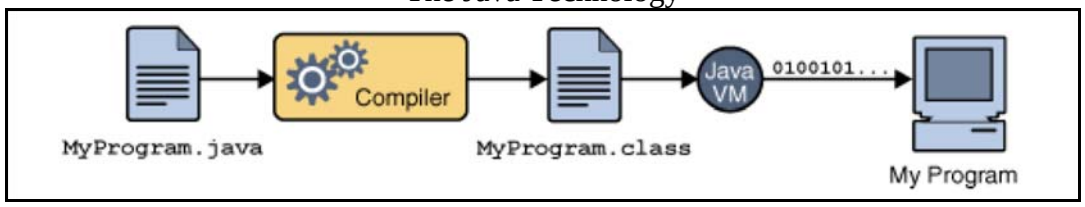
Computer programs were written in human-readable programming languages, such as C and C++. Code written in these human-readable languages — "source code" — was not readable by computer hardware. Only "machine code," which was much simpler, could be used by computers. Most machine code was in a binary language, meaning it consisted entirely of 0s and 1s. Thus, a program had to be converted from source code into machine code before it could be run, or "executed." This conversion traditionally was performed by a "compiler," which converted an entire program in one step and produced an output file of machine code which *then* could be executed. The conversion also could be performed by an "interpreter," which converted portions of the program in different steps, as needed, *while* the program was being executed. No matter how source code was converted to machine code, however, the source code had to be written specifically for the hardware architecture that ultimately would run the machine code. For example, word processing software written for an Apple Mac would not run on an IBM PC, and vice versa.

Java used a "virtual machine" to make software programs more portable. A virtual machine was itself a software program that emulated a particular hardware architecture. Java virtual machines were written for different types of computer hardware (such as Macs and PCs), but they all could read the same type of Java "bytecode." Bytecode was an intermediate form of code that was simpler than source code but not as simple as machine code. Source code for a particular program could be compiled into bytecode instead of machine code, and then the bytecode could be distributed to virtual machines running on top of various types of computer hardware. The virtual machines then could run the program, translating the bytecode into machine code compatible with the particular hardware architecture on which the virtual machine was implemented. Thus, a program written for the Java virtual machine could be run on any

computer with a Java virtual machine, regardless of that computer's underlying hardware architecture.

The public Java tutorials represent this process as follows:

The Java Technology



As depicted above, Java source code was written in the Java programming language, and Java source code was stored in files with the ".java" extension. After the source code was compiled into bytecode, Java bytecode was stored in files with the ".class" extension. The Java virtual machine ("VM") could read the .class files and send instructions to computer hardware in machine-readable binary code. The class files that made up the Java bytecode are important to several of the asserted patents. A basic understanding of object-oriented programming is a prerequisite to understanding Java class files.

Object-oriented programming was a new paradigm that differed from the traditional, "procedural" approach to computer programming. Under the procedural approach, a program typically consisted of a list of commands. The list could be stored in one executable file. Under the object-oriented approach, on the other hand, a program was distributed among various "objects." An object was an encapsulated set of methods and data fields on which the methods could act. Objects could work with one another through defined interfaces. Objects, however, were not *files*. Objects were structures that existed only while a program was running. For purposes of this order, a more detailed description of objects is not necessary; it suffices that objects were the basic components of a running program that was written in an object-oriented programming language.

Objects are relevant, however, in that they are closely related to class files in the Java prior art. In Java, an object was instantiated, or called into being while a program was running, from a single class file. Indeed, multiple objects could be instantiated from the same class file.

This was possible because each Java class file contained the definition of a single class, and each class was a blueprint or prototype from which objects could be created. As noted, Java class files were the files in which the Java bytecode comprising a program was stored. Java class files conformed to the Java class file format, which specified that each class file contained sixteen different portions called "structures" or "items." Each item contained a different type of data or methods. Of the sixteen items that made up a Java class file, only two are relevant to this order: the constant pool table and the methods. The "constant pool" portion of a class file contained a table of constants that were referred to within the class. The "methods" portion of a class file contained a table of all methods declared by the class.

The foregoing discussion sets forth the prior art concepts upon which the asserted patents built. Because bytecode was not as simple as machine code, running bytecode was a more cumbersome task — it consumed more processor time and memory space, both of which were scarce computing resources. Most of the patents Oracle asserts relate to improving the efficiency of this process. Two of the patents relate to improving its security.

Java was developed by Sun Microsystems, Inc. in the 1990s. Oracle Corporation acquired Sun Microsystems and renamed it Oracle America, Inc. in January 2010. Seven months later, Oracle America filed this action claiming infringement of its patents and copyrighted works related to Java improvements. The accused products employ Android, a software platform that was designed for mobile computing devices and that competes with Java in that market. Both Java and Android are complex software platforms. Only part of Java and part of Android are relevant to this action.

One hundred thirty-two claims from the following seven United States patents currently are at issue in this clash of the mobile-computing titans: (1) No. 5,966,702; (2) No. 6,061,520; (3) No. 6,125,447; (4) No. 6,192,476; (5) No. RE38,104; (6) No. 6,910,205; and (7) No. 7,426,720. The parties seek construction of terms and phrases appearing in six of these seven patents. The five terms construed by this order appear in three of them. Overviews of these patents, the disputed terms, and the associated claims are covered in detail in the analysis below.

ANALYSIS

Courts must determine the meaning of disputed claim terms from the perspective of one of ordinary skill in the pertinent art at the time the patent was filed. *Chamberlain Group, Inc. v. Lear Corp.*, 516 F.3d 1331, 1335 (Fed. Cir. 2008). While claim terms "are generally given their ordinary and customary meaning," the "claims themselves provide substantial guidance as to the meaning of particular claim terms." As such, other claims of the patent can be "valuable sources of enlightenment as to the meaning of a claim term." Critically, a patent's specification "is always highly relevant to the claim construction analysis." *Phillips v. AWH Corp.*, 415 F.3d 1303, 1312–15 (Fed. Cir. 2005) (en banc) (internal quotations omitted). Indeed, claims "must be read in view of the specification, of which they are a part." *Markman v. Westview Instruments, Inc.*, 52 F.3d 967, 979 (Fed. Cir. 1995) (en banc), *aff'd*, 517 U.S. 370 (1996). Finally, courts also should consider the patent's prosecution history, which "can often inform the meaning of the claim language by demonstrating how the inventor understood the invention and whether the inventor limited the invention in the course of prosecution, making the claim scope narrower than it would otherwise be." These components of the intrinsic record are the primary resources in properly construing claim terms. Although courts have discretion to consider extrinsic evidence, including dictionaries, scientific treatises, and testimony from experts and inventors, such evidence is "less significant than the intrinsic record in determining the legally operative meaning of claim language." *Phillips*, 415 F.3d at 1317–18 (internal quotations omitted).

While this order acknowledges that the parties have a right to the construction of all disputed claim terms by the time the jury instructions are settled, the Court will reserve the authority, on its own motion, to modify the constructions in this order if further evidence — intrinsic or extrinsic — warrants such a modification. Given that claim construction is not a purely legal matter, but is (as the Supreme Court describes it) a "mongrel practice" with "evidentiary underpinnings," it is entirely appropriate for the Court to adjust its construction of claims prior to trial if the evidence compels an alternative construction. *Markman*, 517 U.S. at 378, 390. The parties should be aware, however, that they are *not* invited to ask for

reconsideration of the constructions herein. Motions for reconsideration may be made only in strict accordance with the rules of procedure, if at all.

1. THE '702 PATENT

The '702 patent, entitled "Method and Apparatus for Pre-Processing and Packaging Class Files," was issued on October 12, 1999. Sun Microsystems is the assignee of the '702 patent; as explained, plaintiff Oracle is the successor to Sun. Nine claims from the '702 patent are asserted in this litigation: independent claims 1, 7, and 13, and dependent claims 5, 6, 11, 12, 15, and 16, which refer to them. One term construed by this order is found in the '720 patent. It is italicized in the claims below.

Claim 1 covers (col. 51:2–12):

> 1. A method of pre-processing class files comprising:
>
> determining plurality of duplicated elements in a plurality of class files;
>
> forming a shared table comprising said plurality of duplicated elements;
>
> removing said duplicated elements from said plurality of class files to obtain a plurality of *reduced class files*; and
>
> forming a multi-class file comprising said plurality of *reduced class files* and said shared table.

Claims 7 covers (col. 51:43–59):

> 7. A computer program product comprising:
>
> a computer usable medium having computer readable program code embodied therein for pre-processing class files, said computer program product comprising:
>
> computer readable program code configured to cause a computer to determine a plurality of duplicated elements in a plurality of class files;
>
> computer readable program code configured to cause a computer to form a shared table comprising said plurality of duplicated elements;
>
> computer readable program code configured to cause a computer to remove said duplicated elements from said plurality of class files to obtain a plurality of *reduced class files*; and

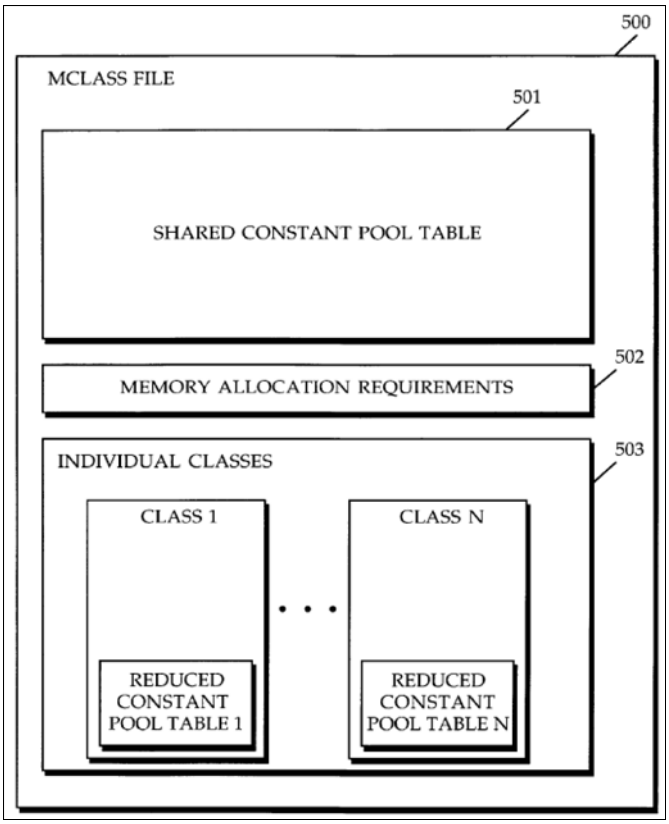
> computer readable program code configured to cause a computer to form a multi-class file comprising said plurality of *reduced class files* and said shared table.

Claim 13 covers (col. 52:39–53):

> 13. An apparatus comprising:
>
> a processor;
>
> a memory coupled to said processor;
>
> a plurality of class files stored in said memory;
>
> a process executing on said processor, said process configured to form a multi-class file comprising:
>
> > a plurality of *reduced class files* obtained from said plurality of class files by removing one or more elements that are duplicated between two or more of said plurality of class files; and
> >
> > a shared table comprising said duplicated elements.

The essence of the invention claimed by the '702 patent is the "multi-class file" format, which is a package containing a set of reduced class files and other related items. One embodiment of the multi-class file format is depicted as follows (col. 9:66–67):

Figure 5 from the '702 Patent: A Multi-Class File



A. "reduced class file"

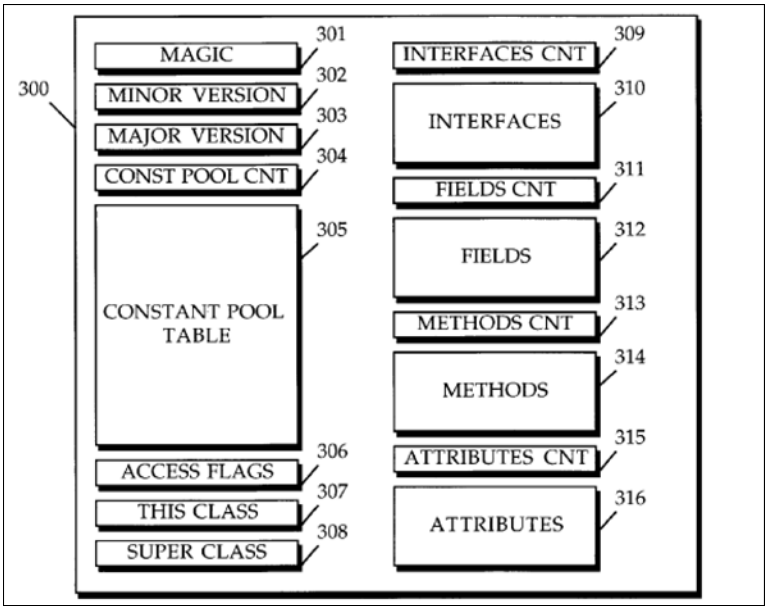
The parties dispute the phrase "reduced class file." The parties define this term differently, but only Google seeks a construction to clarify it for the jury. The parties' proposed constructions are shown below.

ORACLE'S PROPOSED CONSTRUCTION	GOOGLE'S PROPOSED CONSTRUCTION
No construction necessary. A "reduced class file" contains a subset of the code and data contained in a class file.	"a class file containing a subset of the data and instructions contained in a corresponding original class file"

The construction of this phrase is relevant to the parties' infringement arguments. Oracle and Google agree that a "reduced class file" contains a subset of the contents of a "class file," though they characterize the contents differently — "code and data" versus "data and instructions." The main dispute regarding this term, however, is whether a "reduced class file" is a "class file." This order does not find good cause for adopting such a limitation.

The parties disagree as to what a "class file" is, but they did not select that phrase for construction and did not brief the issue. The patent specification uses the phrase "class file" to refer to the Java class file format (cols. 7:23, 11:25). In the prior art, Java compilers compiled Java source code into Java bytecode, which was made up of Java class files. Each Java class file stored the definition of one class (col. 7:31–32). The '702 patent specification incorporated a detailed description of the Java class file format and illustrated it graphically in Figure 3 (col. 7:53).

Figure 3 from the '702 patent: A Java Class File



As shown in Figure 3, a Java class file contained a set of sixteen "structures" or "items," such as the constant pool table (col. 11:48–49). Each item contained a different type of data or methods. As a group, the sixteen items of a Java class file contained information defining a single class. A "class" was a general concept from the object-oriented style of computer programming, as explained previously. On the other hand, a Java class file (which defined a class) had to comply with strict format requirements. The patent specification and the Java literature set forth detailed descriptions of how a Java class file and each of its items must be expressed in order to work with the rest of the Java platform (cols. 7–43). Having described what a class file generally is in the

context of the '702 patent, this order turns to the question of whether a reduced class file must be a class file.

The most that can be gleaned about reduced class files from the claims themselves is that reduced class files are obtained by removing duplicated elements from a plurality of class files (col. 51:7–8, 54–56). Thus, the patent teaches that if one starts with a class file, and performs certain operations on that class file, then one is left with a "reduced class file." The before-item is a class file. The after-item is a reduced class file. The after-item is different from the before-item. There is no indication in the claims, specification, or prosecution history that the after-item must meet the detailed format requirements for Java class files. The after-item is simply whatever is left after the before-item has been "reduced." Presumably, it may or may not qualify as a class file, depending on what specific changes it underwent.

Google argues that the claim language and specification show that a reduced class file is a class file. *First*, Google notes that the phrase "reduced class file" contains the phrase "class file." According to Google, this observation implies that a reduced class file is a class file, just as a green house is a house. *Second*, Google notes that the specification refers to reduced class files as simply "class files" or "classes" in some circumstances (*e.g.*, cols. 4:64–5:5, Fig. 4). According to Google, this second observation implies that the after-item produced by reducing a class file is something that still qualifies as a class file (Google Br. 12–13). This order disagrees with Google on both accounts. The fact that the patent uses the phrases "reduced class file" and "class file" to refer to the after-item that is obtained by reducing a class file is simply an artifact of the drafting process. These phrases indicate the *provenance* of the item produced by the reduction operation. They do not indicate that the essential *features* of that item are identical to those of the item from which it was produced.

By way of analogy, one might start with a document and then shred the document to produced a shredded document. One might go on to describe sweeping up and disposing of "the shredded document" or "the document" even though the item produced by the shredding process might not be considered a document on its own terms. So too here. Just as a document is shredded to produce a shredded document, a class file is reduced to produce a reduced class file.

The use of such convenient terminology is not strong enough evidence to support a finding that the reduced class files disclosed in the '702 patent must satisfy all the format requirements for Java class files.

Because the specification does not define the phrase "reduced class file," this order will construe the phrase in accordance with the limited elaboration provided in the claim language. Independent claims one, seven, and thirteen state that reduced class files are obtained by removing one or more duplicated elements from a plurality of class files that contain the same element or elements. Accordingly, the term "reduced class file" shall be construed as "what remains after one or more duplicated elements have been removed from a class file." This order neither requires nor prohibits reduced class files from qualifying as Java class files.

2. THE '520 PATENT

The '520 patent, entitled "Method and System for Performing Static Initialization," was issued on May 9, 2000. Sun Microsystems is the assignee of the '520 patent, and plaintiff Oracle is the successor to Sun. Twenty-two claims from this patent are asserted in this litigation: independent claims 1, 6, 12, and 18, and dependent claims 2–4, 7–11, 13–17, and 19–23. One of the disputed terms construed by this order is found in the '520 patent. It appears only in dependent claims 3 and 4. It is italicized in the claims below.

Claim 1 covers (col. 9:47–62):

> 1. A method in a data processing system for statically initializing an array, comprising the steps of:
>
> compiling source code containing the array with static values to generate a class fie with a clinit method containing byte codes to statically initialize the array to the static values;
>
> receiving the class file into a preloader;
>
> simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader;
>
> storing into an output file an instruction requesting the static initialization of the array; and
>
> interpreting the instruction by a virtual machine to perform the static initialization of the array.

Claim 3 covers (cols. 9:66–10:5):

> 3. The method of claim 1 wherein *the play executing step* includes the steps of:
>
> allocating a stack;
>
> reading a byte code from the clinit method that manipulates the stack; and
>
> performing the stack manipulation on the allocated stack.

Claim 4 covers (col. 10:6–12):

> 4. The method of claim 1 wherein *the play executing step* includes the steps of:
>
> allocating variables;
>
> reading a byte code from the clinit method that manipulates local variables of the clinit method; and
>
> performing the manipulation of the local variables on the allocated variables.

The ’520 patent, issued seven months after the ’702 patent, provided another way to improve the efficiency of the Java system. Recall that in the basic Java setup, Java source code was compiled by a Java compiler into Java bytecode. The bytecode was contained in class files with the .class extension. The invention of the ’702 patent provided a way to compress the class files and package a group of them into a multi-class (.mclass) file. A Java virtual machine then could run the .mclass file more efficiently than it could have run the set of .class files from which it was made if the .class files had not been pre-processed. A “preloader” performed this pre-processing and packaging of .class files into .mclass files. The invention claimed in the ’520 patent is another pre-processing task to be performed by the preloader, which makes the ultimate task of running the .mclass file on the virtual machine even more efficient.

Whereas the ’702 patent provided a way to compress some of the *data* stored in class files, the ’520 patent provided a way to compress some of the *method* instructions, also stored in class files, that operate on that data. Specifically, the invention of the ’520 patent provided “an improved system for initializing static arrays in the Java programming environment” (col. 3:44–45). Essentially, it “reduc[es] the amount of code executed by the virtual machine to

statically initialize an array" (col. 2:62–64). The Java compiler generated a special method, <clinit>, to perform class initialization, including initialization of static arrays (col. 1:59–61). Under the invention of the '520 patent, when a preloader is consolidating a set of .class files into an .mclass file, the preloader scans the class files for any <clinit> method. The preloader reads any found <clinit> method against memory to see what would happen if the <clinit> method actually were executed, and reduces the result into a set of shorthand instructions, which it stores in the .mclass file as a replacement for the bulkier <clinit> method code (cols. 2:61–3:29, 3:43–4:11). Thus, less code must be loaded and executed by the virtual machine in order to initialize the classes in the multi-class file.

A. "the play executing step"

The parties dispute the term "the play executing step" in dependent claims three and four of the '520 patent. The parties' proposed constructions are shown below.

ORACLE'S PROPOSED CONSTRUCTION	GOOGLE'S PROPOSED CONSTRUCTION
"The play executing step" in claims 3 and 4 is a reference to the "simulating execution" step in claim 1.	Indefinite — cannot be construed.

The construction of this term is relevant to the parties' invalidity arguments. Google argues that the term "the play executing step" in dependent claims three and four is indefinite for lack of an antecedent basis in claim one. Oracle disagrees, arguing instead that the term refers to the "simulating execution" step of claim one. This order agrees with Oracle. Play executing and simulating execution of code are synonymous terms in the '520 patent.

The specification of the '520 patent explicitly defines "simulates executing" and "play executes" as synonyms. Both the summary of the invention and the detailed description of the invention state that the preloader "simulates executing ('play executes')" the <clinit> methods it finds (cols. 2:65–67, 3:59–60). The detailed description further explains: "When play executing, discussed below, the preloader simulates execution of the byte codes contained in the <clinit> method by the virtual machine." (col. 4:64–66). In other portions of the specification, the terms are used interchangeably (cols. 3:12, 3:25, 4:38–39, 6:26–27). In light of the specification, a

person of ordinary skill in the art at the time of the invention would have understood "the play executing step" of claim one to be "simulating executing of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader."

The prosecution history of the '520 patent helps explain why the first set of claims contains this mismatched language. (By contrast, independent claim six and all of its dependent claims refer to "play executing," and independent claim eighteen and all of its dependent claims refer to "simulating execution of" code.) In the original patent application, the third method step of the first claim was drafted as "play executing the byte codes of the clinit method against a memory to identify the static initialization of the array by the preloader" (Peters Supp. Exh. 11 at '520 Application). Claims one and three were rejected as anticipated by a publication that the examiner found to disclose a similar method, including the step of "compar[ing] the execution of byte codes of the clinit method against memory to identify the static initialization of an array by the preloader" (*id.* at Office Action, July 21, 1999). The examiner also objected to claims two, four, and five as dependent upon a rejected base claim.

In response to the rejection of claims one and three, the applicants amended claim one. The phrase "without executing the byte codes" was added, and "play executing" was replaced with "simulating execution of" (*id.* at Response, Oct. 18, 1999). During a telephone interview, the examiner agreed that this amendment to claim one rendered all of the pending claims allowable over the cited art. The examiner's summary of that interview stated that the "general background of the invention was discussed" but that "[n]o prior art was discussed." The examiner wrote: "In particular, the disclosure of the 'pre-loader' and 'simulation' function as disclosed in claims 1, 6 and 12.. [*sic*] Narrowing the claims to make clear the functions of these elements were [*sic*] also discussed." (*id.* at Interview Summary, Oct. 13, 1999). The claims ultimately were allowed with a finding that the prior art "fails to anticipate or render obvious the simulation of execution with respect to class initialization, without executing byte codes." In particular, the examiner found that the reference that prompted the amendment "teaches of

comparing the execution of byte codes, but fails to address the issue of simulating the process without execution" (*id.* at Notice of Allowability).

Thus, the key distinction was that the method step claimed as part of the '520 patent invention did *not* involve actually executing the code. Changing "play executing" to "simulating executing of" may have helped clarify this point, but adding the phrase "without executing the byte codes" is what explicitly narrowed the claim to avoid the prior art. The prior art addressed "comparing execution" but did not use the phrase or concept of "play executing" (*id.* at Notice of Allowability; M. Cierniak, and Wei Li, *Briki: An Optimizing Java Compiler*, proceedings of Compcon conference, Feb. 23–26, 1997). Moreover, the examiner's characterization of application claims one, six, and twelve as each containing a "simulation" function shows that he, too, viewed "play executing" and "simulating execution of" as the same thing; each of those claims recited "play executing," and none of them used any form of the word "simulation." Despite Google's best efforts to show otherwise, there is no evidence in the prosecution history that the applicant surrendered the position that "play executing" and "simulating execution of" are equivalent. The third method step of claim one was reworded from "play executing" to "simulating execution of," and the dependent claims referring "play executing" were not also reworded to match. That is all.

The "play executing" reference in claims three and four may be sloppy, but it is not insolubly indefinite. *See Exxon Research and Eng'g Co. v. United States*, 265 F.3d 1371, 1375 (Fed. Cir. 2001). The specification makes plain that "play executing" is "simulating execution of," and the prosecution history does not compel a contrary interpretation. Indeed, if the examiner believed there was an important difference between "play executing" and "simulating execution of," the patent would not have issued as is. Google's arguments that the disputed term is indefinite and that the prosecution history shows a difference of meaning between "play executing" and "simulating execution of" are refuted by the foregoing analysis. Furthermore, adopting Oracle's proposed construction of "play executing," which is simply a recognition that it refers to the "simulating execution" step of claim one, would not rise to the level of judicial

correction of a claim-drafting error, as Google suggests. *See Novo Indus., L.P. v. Micro Molds Corp.*, 350 F.3d 1348, 1354 (Fed. Cir. 2003).

Having fully considered the parties' arguments, the '520 patent, and its prosecution history, this order adopts Oracle's construction of "the play executing step." Google's vigorous hand-waiving simply does not get around the specification's explicit definition of "play executing" and "simulating execution of" as synonyms. Accordingly, the "the play executing step" in claims three and four will be construed as a reference to the "simulating execution" method step of claim one.

3. THE '104 PATENT

The '104 patent, entitled "Method and Apparatus for Resolving Data References in Generated Code," was issued on April 29, 2003. It is a continuation of U.S. Patent No. RE36,204, which in turn claims priority from U.S. Patent No. 5,367,685, issued on November 22, 1994. Sun Microsystems is the assignee of the '104 patent, and plaintiff Oracle is the successor to Sun. Thirty-one claims from this patent are asserted in this litigation: independent claims 11–13 and 17–41, and dependent claims 14–16. Three of the disputed terms construed by this order are found in the '520 patent. Each of these terms appears in most of the asserted claims. The terms are italicized in the claim below, which is provided as a representative example.

Claim 12 covers (col. 7:14–27):

> 12. A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting *intermediate form object code* comprised of instructions, certain of said instructions containing one or more *symbolic references*, said method comprising the steps of:
>
> interpreting said instructions in accordance with a program execution control; and
>
> *resolving* a *symbolic reference* in an instruction being interpreted, said step of *resolving* said *symbolic reference* including the substeps of:
>
> determining a numerical reference corresponding to said *symbolic reference*, and
>
> storing said numerical reference in a memory.

The claimed invention is "a hybrid compiler-interpreter" "for generating executable code and resolving data references in the generated code" (abstract; cols. 2:27–28, 3:33–34). Under the claimed invention, a compiler compiles source code into an intermediate form of code, and then an interpreter translates the intermediate form code for execution. The intermediate form code contains both numeric and symbolic data references, so the interpreter must employ a different subroutine to handle each type of reference while translating the intermediate form code. Specifically, a static field reference routine is used for numeric references, and a dynamic field reference routine is used for symbolic references (cols. 2:27–59, 3:33–5:57).

A. "intermediate form code" and "intermediate form object code"

The parties agree that "intermediate form code" and "intermediate form object code" are the same thing in the context of the '104 patent, but they disagree as to what these terms mean. These disputed terms appear in about two-thirds of the thirty-one asserted claims of the '104 patent. The parties' proposed constructions are shown below.

ORACLE'S PROPOSED CONSTRUCTION	GOOGLE'S PROPOSED CONSTRUCTION
"executable code that is generated by compiling source code and is independent of any computer instruction set"	"code that is generated by compiling source code and is independent of any computer instruction set"

The construction of these terms is relevant to the parties' invalidity arguments. The parties agree that "intermediate form code" and "intermediate form object code" are "code that is generated by compiling source code and is independent of any computer instruction set." The parties' sole disagreement with respect to these terms is whether the intermediate code must be executable. This order finds that it does. Most persuasive is the following statement, which appears twice in the specification, introducing both the summary of the invention and the detailed description of presently preferred and alternate embodiments: "A method and apparatus for generating executable code and resolving data references in the generated code is disclosed" (cols. 2:27–29, 3:33–35). That is, executable code is generated, and data references in the generated excutable code are resolved. In every claimed embodiment of the invention that uses the disputed terms, the "intermediate form code" or "intermediate form object code" is the code that has been generated

by a compiler and that contains data references which are then resolved by an interpreter. Hence, the "intermediate form code" or "intermediate form object code" in these claims is the "generated code" of the invention, which the specification explicitly states is executable.

Other intrinsic evidence supports this conclusion as well. For example, the specification notes that "generated coded [*sic*] are in intermediate form" (col. 4:38–39). This statement confirms that the "intermediate form code" and "intermediate form object code" in the claims are the "generated code" that the specification describes as executable. The specification also refers to the "execution performance" of "the 'compiled' intermediate form object code" of the invention, suggesting not only that it is executable, but also that it can be executed as efficiently as "traditional compiled object code" while maintaining greater flexibility (col. 5:41–49). Furthermore, the background portion of the specification lays the foundation that a "program in intermediate form *is executed* by" an interpreter (cols. 1:67–2:2) (emphasis added).

The claims themselves also refer to the intermediate code as executable. Asserted independent claim eleven, for example, claims "intermediate form object code constituted by a set of instructions, certain of said instructions containing one or more symbolic references," as well as "a processor configured *to execute* said instructions containing one or more symbolic references" (col. 7:6–10) (emphasis added). Google protests that the processor of claim eleven executes "instructions" rather than "intermediate form object code," but the plain language of the claim explains that these instructions *are* the intermediate form object code. The other asserted claims have similar references to execution.

Independent claims one and six, which were not re-issued, specifically claim "generating executable code in an intermediate form" (cols. 5:63, 6:36–37). Oracle views this claim language as further support for the position that code in intermediate form is executable. Google, on the other hand, views the exclusion of these claims from the re-issued '104 patent as proof of "the patentee's express intent to claim 'intermediate form [object] code' that is not necessarily executable" (Google Resp. Br. 16). Google, however, does not point to any evidence from the prosecution history that would support this inference of patentee intent.

Having reviewed the prosecution history of the '104 patent, this order finds no support for such an inference. The applicant explained that he sought re-issuance because the original patent claimed less than he had a right to claim in three respects, none of which had anything to do with expanding the scope of the claims to include intermediate form code that is not executable. Rather, the expansions of the claims were intended to cover the following three areas: (1) complimentary compilers and interpreters that could be sold separately rather than packaged into a single product; (2) more specific "handling of numerical values corresponding to symbolic references"; and (3) *Beauregard*-type "claims more specifically directed to computer program code devices." (Peters Supp. Exh. 11 at '104 Gosling Decl., Nov. 21, 1996). Moreover, the '104 patent was issued after the applicant overcame a series of rejections, but all the rejections were based on procedural defects, not prior art. The applicant never narrowed the scope of the claims to avoid prior art, nor did he affirmatively disclaim intermediate form code that was or was not executable (*see id.* at '104 office actions and responses).

Google also argues for its inference of patentee intent as a matter of law. Since claims one and six refer to the intermediate code as "executable" but the other claims do not, Google infers that the property of being executable is not a limitation of the intermediate code in the other claims. *See Philips*, 415 F.3d at 1314. Drawing this inference may be proper as a matter of law, but the inference is overcome by the strong countervailing evidence from the specification discussed above.

Google's remaining arguments are unpersuasive as well. The critique that Oracle's proposed construction "would limit the term to code that is not an 'intermediate representation'" (Google Resp. Br. 17) raises no concern at all, because the patent claims and specification repeatedly make clear that "intemediate form code" and "intermediate representations" of code are two different things (*e.g.*, col. 4:21–23; Figs. 4–5; clm. 21). Similarly, the fact that the limitation "executable" could encompass both execution directly by a computer processor (hardware) and execution by a virtual machine (software) is neither confusing nor problematic.

Although the limitation that "intermediate form code" and "intermediate form object code" must be executable is not explicitly called out in the asserted claims, this order finds that

the limitation would be implicit to a person of ordinary skill in the art reading the claims in light of the surrounding claim language and specification. Accordingly, Oracle's proposed construction of these terms will be adopted. Both "intermediate form code" and "intermediate form object code" will be construed to mean "executable code that is generated by compiling source code and is independent of any computer instruction set."

B. "symbolic reference"

The parties disagree both as to whether the term "symbolic reference" requires construction, and, if so, how it should be construed. This disputed term appears in each of the thirty-one asserted claims of the '104 patent. The parties' proposed constructions are shown below.

ORACLE'S PROPOSED CONSTRUCTION	GOOGLE'S PROPOSED CONSTRUCTION
No construction necessary. The ordinary meaning is "a reference by name."	"a dynamic reference to data that is string- or character-based"

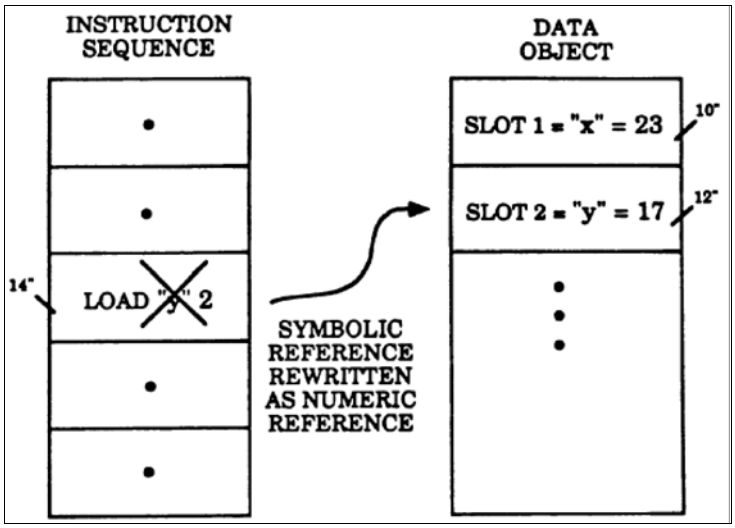
The construction of this term is relevant to the parties' infringement arguments. Oracle asserts that the ordinary meaning of "symbolic reference" is "a reference by name," whereas Google would construe "symbolic reference" to mean a data reference that is dynamic and is string- or character-based. A person of ordinary skill in the art might find both parties' proposed constructions to be technically accurate, but they rely heavily on extrinsic evidence and would not be particularly helpful to a jury. Instead of choosing between them, this order steers a middle course guided by the intrinsic evidence of the '104 patent itself.

The '104 patent teaches two different types of data references: numeric references and symbolic references. The claimed invention includes an interpreter with two different sub-routines: "a static field reference routine for handling numeric references and a dynamic field reference routine for handling symbolic references" (col. 2:42–44). Computers stored data in memory locations defined by numeric addresses. A numeric data reference was one that identified data directly by its memory-location address. For example, the command "load the data stored in memory slot 2" contains a numeric reference to the data stored in slot 2 (col.

1:26–41). The claimed invention would use a static subroutine to interpret this numeric data reference — all it would have to do is go get whatever data is stored in slot 2. The data stored in slot 2 might turn out to be, for example, "17" (col. 5:24–31).

A symbolic data reference, on the other hand, did *not* identify data directly by its memory-location address. Instead, a symbolic reference identified data by a "symbolic name" (col. 1:64–67). For example, the command "load the data called y" contains a symbolic reference to the data called y. The claimed invention would use a dynamic subroutine to interpret this symbolic reference — it would have to figure out that "y" means "17" or that "y" means "the data stored in memory slot 2," and *then* get the data called y (col. 5:13–19). Figure 8 depicts the step of rewriting a symbolic reference as a numeric reference, which is included in some embodiments of the invention.

Figure 8 from the '104 Patent:
Rewriting a Symbolic Reference as a Numeric Reference



This contrast between symbolic references and numeric references pervades the asserted claims. Claim twelve, which is reproduced above, discloses "resolving a symbolic reference" to include the substeps of "determining a numerical reference corresponding to said symbolic reference, and storing said numerical reference in a memory" (col. 7:33–37). Many of the other claims contain similar language.

The numeric and symbolic references discussed in the '104 patent are references to data. The patent discloses these two types of data references in opposition to one another, and the specification refers to symbolic data references as references made by a "symbolic name." These observations from the intrinsic record shall guide construction of the disputed term. The patent's distinction that symbolic references are resolved in a dynamic manner, whereas numeric references are resolved in a static manner, provides a further gloss that would be useful to a jury. On the other hand, delving into extrinsic dictionaries to construe the term "symbolic reference" is unnecessary. Google's proposed modifier "string- or character-based" does not correspond to any terms or concepts appearing in the intrinsic record and will not be read in from the proffered extrinsic sources.

In its critique of the tentative claim-construction order, Oracle counsels against adopting a gloss that refers to dynamic resolution. *First*, Oracle cites intrinsic evidence it interprets as showing that symbolic references "need not be resolved dynamically" (Dkt. No. 132 at 1–2). This evidence concerns only the prior art. Although the abstract concept of a symbolic reference may not require dynamic resolution, the concrete invention claimed by the '104 patent resolves symbolic references dynamically. Thus, for purposes of any infringement or invalidity analysis in this action, the dynamic-resolution gloss is apt. *Second*, Oracle voices concern that the word "dynamic" has "many nuanced meanings that depend on its use in context" (*id.* at 2). Because this word comes directly from the '104 patent, its use therein will further inform the construction of "symbolic reference." The word "dynamic" is not being imported from a vacuum. *Third*, Oracle points out that "Google's programmers wrote that Android 'converts symbolic references into pointers,' using the same language that the patent does." Oracle warns that construing the term "symbolic reference" to require dynamic resolution might help Google "slip the noose of its own creation" by arguing that its symbolic references do not infringe the symbolic references disclosed in the patent (*ibid.*). This is not a compelling reason to adopt or forego an accurate claim construction. The term "symbolic reference" shall be construed as "a reference that identifies data by a name other than the numeric memory location of the data, and that is resolved dynamically rather than statically."

C. "resolve" and "resolving"

As with the term "symbolic reference," the parties disagree both as to whether the term "resolve" or "resolving" requires construction, and, if so, how it should be construed. This disputed term appears in the vast majority of the thirty-one asserted claims of the '104 patent. The parties' proposed constructions are shown below.

ORACLE'S PROPOSED CONSTRUCTION	GOOGLE'S PROPOSED CONSTRUCTION
No construction necessary. "Resolving" a symbolic reference is determining its corresponding numerical reference.	"replace/replacing at least for the life of the process"

The construction of this term is relevant to the parties' infringement arguments. The parties disagree substantially, both as to what sort of action resolving is (*i.e.*, determining versus replacing) and whether this action carries any temporal limitation within the context of the '104 patent. Again, this order finds that Google's attempt to pile on limitations is not supported by the intrinsic evidence.

The asserted claims disclose resolving symbolic data references. As an initial matter, this order notes that the term "resolve" is used only in conjunction with symbolic references in the '104 patent; numeric references are not "resolved." Accordingly, the term "resolve" shall be construed within the limited context of resolving symbolic data references for purposes of the '104 patent. Some asserted claims provide that resolving a symbolic reference is accomplished "by determining a numerical reference corresponding to said symbolic reference, and storing said numerical reference in a memory" (*e.g.*, clm. 19). Other asserted claims define resolving a symbolic reference to include "obtaining data in accordance to said stored numerical reference" as well (*e.g.*, clm. 18). Sill other asserted claims refer to "resolving said symbolic references to corresponding numerical references" or "resolv[ing] the symbolic data reference" without any further explanation (clms. 17, 24). Such claims, however, also generally refer to "a numeric reference resulting from the resolution of the symbolic reference."

Based on the extensive usage of "resolve" in the asserted claims, this order concludes that "resolving" a symbolic data reference requires at least identifying the corresponding numeric

reference by which the data can be referenced. Resolving a symbolic reference does not, however, require *replacing* the symbolic reference. Specifically, the fact that a symbolic reference may be resolved "by determining a numerical reference corresponding to said symbolic reference, and storing said numerical reference in *a* memory" (emphasis added) makes plain that the resolution process can be completed for a symbolic reference without writing over the symbolic reference. Furthermore, some claims use the words "resolve" and "replace" to refer to two different steps. For example, claim 30 recites "replacing" instructions containing symbolic references "with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routine to resolve the symbolic data reference" (col. 10:9–13). Similarly, claims one and six (which were not reissued) disclose "replacing said symbolic references with their corresponding numeric references" as a separate step to be performed *after* the step of "resolving said symbolic references to corresponding numeric references" is complete (col. 6:3–5, 44–47).

The specification and the prosecution history are consistent with this claims-based interpretation. Google leans heavily on the aspect of the invention that contemplates resolving each symbolic reference only once per execution run (Google Br. 17–21). This improvement, however, does *not* require the resolution of symbolic references to include *replacing* them. The advantage could be achieved instead by storing the corresponding numeric references in another convenient location. Indeed, the patent claims both approaches (*e.g.*, clms. 13 (storing), 30 (replacing)). The storage or replacement also need not persist "for the life of the process" as Google suggests. Nothing in the intrinsic record addresses a temporal aspect of "resolving." Even to practice the single-resolution improvement on which Google relies, the particulars of the process in question would dictate the length of time the resolution of a particular symbolic reference would need to last. Conceivably, if a lengthy process utilized a particular symbolic reference only early on, it might be beneficial to store the corresponding numeric reference in memory during the first part of the process but then empty and re-use that memory space as the process continues to run. The '104 patent does not foreclose this possibility. The term "resolve" should not be so limited. Having considered the parties' arguments in the context of the intrinsic

record, this order holds that "resolving" a symbolic reference shall be construed to mean "at least determining the numerical memory-location reference that corresponds to the symbolic reference."

4. REMAINING TERMS AND PATENTS

In an attempt to engineer a three-for-one deal or better, the parties seek construction of the following set of terms: "computer-usable medium," which appears in the '702 patent, "computer-readable storage medium," which appears in the '720 patent, and "computer-readable medium," which appears in the '104, '520, '447, and '476 patents. These three terms come from six different patents whose issue dates span a decade and whose subject matter ranges from security and access protections to loading and processing techniques.

Claim terms are to be construed from the perspective of one of ordinary skill in the pertinent art at the time of filing. *Chamberlain*, 516 F.3d at 1335. These six patents relate to different specialized arts within the field of computer science — a field that is known for rapid change. Construing these terms properly would require individualized attention to the intrinsic evidence and prosecution history of each of the six patents from which they hail.

The parties were instructed to isolate no more than six disputed terms for construction by way of the *Markman* proceedings (Dkt. No. 56 ¶ 5). This order already has construed five. The request for construction of this last set of terms, which is equivalent to at least three and as many as six additional terms, is too much. No more terms will be construed at this time.

CONCLUSION

For the reasons provided herein, the constructions set forth above will apply in this dispute. The Court will reserve the authority, on its own motion, to modify these constructions if further evidence warrants such a modification. Counsel, however, may not ask for modification.

IT IS SO ORDERED.

Dated: May 9, 2010.

WILLIAM ALSUP
UNITED STATES DISTRICT JUDGE